Generic Encodings of Constructor Rewriting Systems

Horatiu Cirstea, Pierre-Etienne Moreau

Generic Encodings of Constructor Rewriting Systems

Horatiu Cirstea
Université de Lorraine – LORIA
Horatiu.Cirstea@loria.fr

Pierre-Etienne Moreau
Université de Lorraine – LORIA
Pierre-Etienne.Moreau@loria.fr

Rewriting is a formalism widely used in computer science and mathematical logic. The classical formalism has been extended, in the context of functional languages, with an order over the rules and, in the context of rewrite based languages, with the negation over patterns. We propose in this paper a concise and clear algorithm computing the difference over patterns which can be used to define generic encodings of constructor term rewriting systems with negation and order into classical term rewriting systems. As a direct consequence, established methods used for term rewriting systems can be applied to analyze properties of the extended systems. The approach can also be seen as a generic compiler which targets any language providing basic pattern matching primitives. The formalism provides also a new method for deciding if a set of patterns subsumes a given pattern and thus, for checking the presence of useless patterns or the completeness of a set of patterns.

1 Introduction

Rewriting is a very powerful tool used in theoretical studies as well as for practical implementations. It is used, for example, in semantics in order to describe the meaning of programming languages, but also in automated reasoning when describing by inference rules a logic, a theorem prover or a constraint solver. It is also used to compute in systems making the notion of rule an explicit and first class object, like Mathematica [15], Maude [6], or Tom [3]. Rewrite rules, the core concept in rewriting, consist of a pattern that describes a schematic situation and the transformation that should be applied in that particular case. The pattern expresses a potentially infinite number of instances and the application of the rewrite rule is decided locally using a (matching) algorithm which only depends on the pattern and its subject.

Comparing to the general rewriting formalism where rule application is decided locally and independently of the other rules, rule-based and functional programming languages generally use an order over the rules. This is not only convenient from the implementation point of view but it also allows more concise and clear specifications in some specific cases. In particular, this order might avoid an exhaustive specification of alternative and default cases. For instance, if we consider a term representation of motor vehicles we can use the following list of rules

$$
\begin{array}{l}
[\ paint(car(x,suv)) \twoheadrightarrow red,\\
\ \ paint(car(electric,x)) \twoheadrightarrow blue,\\
\ \ paint(car(diesel,y)) \twoheadrightarrow red,\\
\ \ paint(car(x,y)) \twoheadrightarrow white,\\
\ \ paint(x) \twoheadrightarrow red\ \ ]
\end{array}
$$

for the assignment of an imaginary eco-label: all electric cars but the SUVs (which are red) are blue, diesel cars are red and the remaining cars are white; all the other vehicles are red.

Patterns express positive conditions and we have used the term $car(electric,x)$ to specify electric cars of any style. Negation is nevertheless intrinsic to human thinking and most of the time when searching

Submitted

for something, we base our patterns on both positive and negative conditions. We would like for example to specify that we search for all cars that are not SUVs, or for all cars which are neither SUV nor diesel. The notion of pattern has been extended to the one of anti-pattern [10], i.e. patterns that may contain complement symbols, and implemented in tools featuring pattern matching like Tom [4] and Mathematica [15]. With such an approach the above statements can be easily expressed as $car(x, !suv)$ and respectively $car(!diesel, !suv)$, and the eco-labeling can be expressed by the following list of rules with anti-patterns

$$\begin{aligned}
[\ & paint(car(electric, !suv)) \twoheadrightarrow blue, \\
& paint(car(!diesel, !suv))) \twoheadrightarrow white, \\
& paint(x) \twoheadrightarrow red\ \ ]
\end{aligned}$$

Similarly to plain term rewriting systems (TRS), *i.e.* TRS without anti-patterns and ordered rules, it is interesting to analyze the extended systems *w.r.t.* to their confluence, termination and reachability properties, for example. Generally, well-established techniques and (automatic) tools used in the plain case cannot be applied directly in the general case. There have been several works in the context of functional programming like, for example [13, 9, 8, 1] to cite only a few, but they are essentially focused on powerful techniques for analyzing the termination and complexity of functional programs with ordered matching statements. We are interested here in a transformation approach which can be used as an add-on for well-established analyzing techniques and tools but also as a generic compiler for ordered TRS involving anti-patterns which could be easily integrated in any language providing rewrite rules, or at least pattern matching primitives. For example, if we consider trucks and cars with 4 fuel types and 3 styles the transformation we propose will provide the following order independent set of rules:

$$\begin{aligned}
\{\ & paint(car(electric, sedan)) \twoheadrightarrow blue, \\
& paint(car(electric, minivan)) \twoheadrightarrow blue, \\
& paint(car(hybrid, sedan)) \twoheadrightarrow white, \\
& paint(car(hybrid, minivan)) \twoheadrightarrow white, \\
& paint(car(gas, sedan)) \twoheadrightarrow white, \\
& paint(car(gas, minivan)) \twoheadrightarrow white, \\
& paint(truck(x, y)) \twoheadrightarrow red, \\
& paint(car(x, suv)) \twoheadrightarrow red, \\
& paint(car(diesel, x)) \twoheadrightarrow red \quad \}
\end{aligned}$$

for the previous list of rules.

In this paper we propose an extended matching and rewriting formalism which strongly relies on the newly introduced operation of relative complement, and we provide an algorithm which computes for a given difference of patterns $p_1 \setminus p_2$ the set of patterns which match all terms matched by p_1 but those matched by p_2. The algorithm defined itself by rewriting in a concise and clear way turns out to be not only easy to implement but also very powerful since it has several direct applications:

- it can be used to transform an ordered constructor TRS into a plain constructor TRS defining exactly the same relation over terms;
- it can be used to transform an anti-pattern into a set of equivalent patterns and provides thus a way to compile such patterns and to prove, using existing techniques, properties of anti-patterns and of the corresponding rewriting systems;
- it can be used to decide whether a pattern is subsumed by a given set of patterns and thus, to check the presence of useless patterns or the completeness of a set of patterns.

The paper is organized as follows. The next section introduces the notions of pattern, pattern semantics and rewriting system. Section 3 presents the translation of extended patterns into plain patterns and explains how this can be used to detect useless patterns. In Section 4 we present a new technique for eliminating redundant patterns and Section 5 describes the transformation of ordered CTRS involving anti-patterns into plain CTRS. Section 6 presents some optimizations and implementation details. In Section 7 we discuss some related works. We end with conclusions and further work.

2 Pattern semantics and term rewriting systems

We define in this section most of the notions and notations necessary in the rest of the paper.

2.1 Term rewriting systems

We first briefly recall basic notions concerning first order terms and term rewriting systems; more details can be found in [2, 18].

A *signature* Σ consists in an alphabet $\mathcal{F}$ of symbols together with an application *ar* which associates to any symbol f its *arity* (we write $\mathcal{F}^n$ for the subset of symbols of arity n). Symbols in $\mathcal{F}^0$ are called *constants*. Given a countable set $\mathcal{X}$ of *variable* symbols, the set of *terms* $\mathcal{T}(\mathcal{F},\mathcal{X})$ is the smallest set containing $\mathcal{X}$ and such that $f(t_1,\ldots,t_n)$ is in $\mathcal{T}(\mathcal{F},\mathcal{X})$ whenever $f \in \mathcal{F}^n$ and $t_i \in \mathcal{T}(\mathcal{F},\mathcal{X})$ for $i \in [1,n]$.

A *position* of a term t is a finite sequence of positive integers describing the path from the root of t to the root of the sub-term at that position. The empty sequence representing the root position is denoted by ε. $t_{|\omega}$, resp. $t(\omega)$, denotes the sub-term of t, resp. the symbol of t, at position ω. We denote by $t\,[s]_\omega$ the term t with the sub-term at position ω replaced by s. $\mathcal{P}os(t)$ is called the set of positions of t. We write $\omega_1 < \omega_2$ if ω_2 extends ω_1, that is, if $\omega_2 = \omega_1.\omega_1'$ for some non empty sequence ω_1'. We have thus, $\varepsilon < \varepsilon.1$ and $\varepsilon.1 < \varepsilon.1.2$. Notice that $\forall \omega_1, \omega_2 \in \mathcal{P}os(t)$, $\omega_1 < \omega_2$ iff $t_{|\omega_2}$ is a sub-term of $t_{|\omega_1}$.

The set of variables occuring in $t \in \mathcal{T}(\mathcal{F},\mathcal{X})$ is denoted by $\mathcal{V}ar(t)$. If $\mathcal{V}ar(t)$ is empty, t is called a *ground* term. $\mathcal{T}(\mathcal{F})$ denotes the set of all ground terms. A *linear* term is a term where every variable occurs at most once.

We call *substitution* any mapping from $\mathcal{X}$ to $\mathcal{T}(\mathcal{F},\mathcal{X})$ which is the identity except over a finite set of variables $\mathcal{D}om(\sigma)$ called *domain* of σ. A substitution σ extends as expected to an endomorphism σ' of $\mathcal{T}(\mathcal{F},\mathcal{X})$. To simplify the notations, we do not make the distinction between σ and σ'. σ is often denoted by $\{x \mapsto \sigma(x) \mid x \in \mathcal{D}om(\sigma)\}$.

A *rewrite rule* (over Σ) is a pair $(l,r) \in \mathcal{T}(\mathcal{F},\mathcal{X}) \times \mathcal{T}(\mathcal{F},\mathcal{X})$ (also denoted $l \Rightarrow r$) such that $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$ and a *term rewriting system* (TRS) is a set of rewrite rules $\mathcal{R}$ inducing a *rewriting relation* over $\mathcal{T}(\mathcal{F})$, denoted by $\Longrightarrow_{\mathcal{R}}$ and such that $t \Longrightarrow_{\mathcal{R}} t'$ iff there exist $l \Rightarrow r \in \mathcal{R}$, $\omega \in \mathcal{P}os(t)$, and a substitution σ such that $t_{|\omega} = \sigma(l)$ and $t' = t\,[\sigma(r)]_\omega$. The reflexive and transitive closure of $\Longrightarrow_{\mathcal{R}}$ is denoted by $\Longrightarrow_{\mathcal{R}}^*$.

A rewriting system $\mathcal{R}$ is *left-linear* if the left-hand sides of all its rewrite rules are linear. $\mathcal{R}$ is *confluent* when for any terms t, t_1, t_2 *s.t.* $t \Longrightarrow_R^* t_1$ and $t \Longrightarrow_R^* t_2$ there exists a term u *s.t.* $t_1 \Longrightarrow_R^* u$ and $t_2 \Longrightarrow_R^* u$. $\mathcal{R}$ is *terminating* if there exists no infinite rewrite sequence $t_1 \Longrightarrow_{\mathcal{R}} t_2 \Longrightarrow_{\mathcal{R}} \cdots$. A terminating and confluent rewriting system $\mathcal{R}$ is called convergent; for such systems the normal form of t is denoted $t \downarrow_{\mathcal{R}}$.

For the purpose of presenting function definitions with an ML-style pattern matching we consider that the set of symbols $\mathcal{F}$ of a signature is partitioned into a set $\mathcal{D}$ of *defined symbols* and a set $\mathcal{C}$ of *constructors*. The linear terms over the constructor signature $\mathcal{T}(\mathcal{C},\mathcal{X})$ are called *constructor patterns* and the ground constructor patterns in $\mathcal{T}(\mathcal{C})$ are called *value*s. A constructor TRS (CTRS) is a TRS whose rules have a left-hand side of the form $\varphi(l_1,\ldots,l_n) \twoheadrightarrow r$ with $\varphi \in \mathcal{D}^n$ and $l_i \in \mathcal{T}(\mathcal{C},\mathcal{X})$.

2.2 Patterns and their ground semantics

The definition of a function φ by a list of oriented equations of the form:

$$
\begin{array}{lll}
[\ \varphi(p^1_1,\ldots,p^1_n) & \twoheadrightarrow & t^1 \\
 & \vdots & \\
\varphi(p^m_1,\ldots,p^m_n) & \twoheadrightarrow & t^m\ ]
\end{array}
$$

corresponds thus to an ordered CTRS with $\varphi \in \mathcal{D}^n$, $p^j_i \in \mathcal{T}(\mathcal{C},\mathcal{X})$, $t^j \in \mathcal{T}(\mathcal{F},\mathcal{X})$.

When focusing on the underlying pattern matching for such functional specifications the defined symbol in the left-hand side of the equations only indicates the name of the defined function and only the constructor terms are relevant for its definition. We assume thus a set $\mathcal{L} = \{\cdot_1,\ldots,\cdot_n\}$ of suitable symbols for n-tuples (the cardinality of $\mathcal{L}$ is the maximum arity of the symbols in $\mathcal{D}$), and for simplicity an n-tuple $\cdot_n(p_1,\ldots,p_n)$ is denoted $(\!|p_1,\ldots,p_n|\!)$. In order to address the underlying pattern matching of a function definition of the above form we consider the list of tuples of patterns:

$$
\begin{array}{c}
(\!|p^1_1,\ldots,p^1_n|\!) \\
\vdots \\
(\!|p^m_1,\ldots,p^m_n|\!)
\end{array}
$$

All the tuples of patterns $(\!|p_1,\ldots,p_n|\!)$ considered in this paper are linear, *i.e.* each p_i is linear, and a variable can appear in only one pattern p_i. In what follows, we call constructor pattern a constructor pattern or a tuple of constructor patterns. We may use the notation $\overrightarrow{p}$ to denote explicitly a tuple of constructor patterns. Similarly, we call value a term in $\mathcal{T}(\mathcal{C})$ or a tuple of such values and we use the notation $\overrightarrow{v}$ to denote explicitly tuples of values. We also write $\varphi(\overrightarrow{p})$ to denote a term $\varphi(p_1,\ldots,p_n)$, $\varphi \in \mathcal{D}^n$, when there is no need to make explicit the terms $p_1,\ldots,p_n$ in a given context.

Let v be a value and p be a constructor pattern (*i.e.* a constructor pattern or a tuple of constructor patterns), we say that v is an instance of p when there exists a substitution σ (extended to the notion of tuples) such that $v = \sigma(p)$ and in this case we say that *p matches v*. Since p is linear the instance relation can be defined inductively:

$$
\begin{array}{rcll}
x & \prec\!\!\prec & v & x \in \mathcal{X} \\
c(p_1,\ldots,p_n) & \prec\!\!\prec & c(v_1,\ldots,v_n) & \text{iff } \wedge^n_{i=1}\, p_i \prec\!\!\prec v_i, c \in \mathcal{C} \cup \mathcal{L}
\end{array}
$$

Given a list of patterns $P = [p_1,\ldots,p_n]$ we say that P matches a value v with pattern p_i, denoted $P \prec\!\!\prec_i v$, iff the following conditions hold:

$$
\begin{array}{l}
p_i \prec\!\!\prec v \\
p_j \not\prec\!\!\prec v, \quad \forall j < i
\end{array}
$$

Note that if $P \prec\!\!\prec_i v$ then for all $j \neq i$, $P \not\prec\!\!\prec_j v$.

Several pattern matching properties can be expressed in this context [14]:

- a list of patterns P is *exhaustive* iff for all values v there exists an i such that $P \prec\!\!\prec_i v$,
- a pattern $p_i \in P$ is *useless* iff there does not exist a value v such that $P \prec\!\!\prec_i v$.

Starting from the observation that a pattern can be interpreted as the set of its instances we define the semantics of (lists of) patterns and state the relationship to pattern matching.

The *ground semantics* of a constructor pattern $p \in \mathcal{T}(\mathcal{C},\mathcal{X})$ is the set of all its ground constructor instances: $[\![p]\!] = \{\sigma(p) \mid \sigma(p) \in \mathcal{T}(\mathcal{C})\}$. This extends as expected to tuples of constructor patterns: $[\![(\!|p_1,\ldots,p_n|\!)]\!] = \{(\!|\sigma(p_1),\ldots,\sigma(p_n)|\!) \mid \sigma(p_1),\ldots,\sigma(p_n) \in \mathcal{T}(\mathcal{C})\}$. Note that the ground semantics of a variable x is the set of all possible ground patterns: $[\![x]\!] = \mathcal{T}(\mathcal{C})$, and since patterns are linear we can use a recursive definition for the non variable patterns:

$$[\![c(p_1,\ldots,p_n)]\!] = \{c(t_1,\ldots,t_n) \mid (t_1,\ldots,t_n) \in [\![p_1]\!] \times \ldots \times [\![p_n]\!]\},$$

for all $c \in \mathcal{C} \cup \mathcal{L}$.

Proposition 2.1 (Instance relation vs. ground semantics)**.** *Given a pattern p and a value v, $v \in [\![p]\!]$ iff $p \prec\!\!\prec v$.*

The semantics of a set of patterns $P = \{p_1,\ldots,p_n\}$ or of a list of patterns $P = [p_1,\ldots,p_n]$ is the union of the semantics of each of the patterns: $[\![P]\!] = \bigcup_{i=1}^{n} [\![p_i]\!]$. Note that given a value v, $v \in [\![P]\!]$ iff there exists $p_i \in P$ s.t. $p_i \prec\!\!\prec v$. We say that a set of patterns P *subsumes* a pattern p iff $[\![p]\!] \subseteq [\![P]\!]$.

Given a list of patterns $P = [p_1,\ldots,p_n]$, the *disambiguation* problem [12] consists in finding sets of patterns $P_1,\ldots,P_n$ such that for each $i \in [1..n]$, $[\![P_i]\!] = [\![p_i]\!] \setminus \cup_{j=1}^{i-1} [\![p_j]\!]$. Supposing the disambiguation problem can be solved, we have that for any value v, $v \in [\![P_i]\!]$ iff $P \prec\!\!\prec_i v$. Consequently, the definition of a function by a list of equations can be replaced by an equivalent one consisting of a set of equations, *i.e.* one where the order of equations is not important.

The aforementioned properties of pattern matching can be also expressed in terms of ground semantics. Checking the exhaustiveness of a list of patterns $P = [p_1,\ldots,p_n]$ consists in checking whether for any value v there exists an i s.t. $v \in [\![p_i]\!] \setminus \cup_{j=1}^{i-1} [\![p_j]\!]$. Checking if the pattern p_i is a useless case (*w.r.t.* $p_1,\ldots,p_{i-1}$) consists in checking if there exists no value v s.t. $v \in [\![p_i]\!] \setminus \cup_{j=1}^{i-1} [\![p_j]\!]$, *i.e.* checking whether $\{p_1,\ldots,p_{i-1}\}$ subsumes p_i or not. For the latter it is equivalent to check that $[\![p_i]\!] \setminus \cup_{j=1}^{i-1} [\![p_j]\!]$ is empty and for the former it is equivalent to check that $[\![x]\!] \setminus \cup_{j=1}^{n} [\![p_j]\!]$ is empty.

We will come back to the use of disambiguation for generating equivalent function definitions and detecting possible pattern matching anomalies and for now we focus on solving the disambiguation problem. To handle this problem we first define *extended patterns* as follows:

$$p \quad := \quad \mathcal{X} \mid c(p_1,\ldots,p_n) \mid p_1 + p_2 \mid p_1 \setminus p_2 \mid \bot \quad \textit{with } c \in \mathcal{C}$$

Intuitively, a pattern $p_1 + p_2$ matches any term matched by one of its components. The relative *complement* of p_2 *w.r.t.* p_1, $p_1 \setminus p_2$, matches all terms matched by p_1 but those matched by p_2. $\bot$ matches no term. $\setminus$ has a higher priority than $+$. If an extended pattern contains no $\setminus$ it is called *additive* and, if it contains no symbol $\bot$ is called *pure*.

The pattern $p_1 + p_2$ is linear if each of p_1 and p_2 is linear; this corresponds to the fact that p_1 and p_2 represent independent alternatives and thus, that their variables are unrelated *w.r.t.* pattern semantics. For example, the terms $h(x)+g(x)$ and $h(x)+g(y)$ both represent all terms rooted by h or g. An extended pattern of the form $c(p_1,\ldots,p_n)$ is linear if each p_i, $i \in [1..n]$, is linear and $\cap_{i=1}^{n} \mathcal{V}ar(p_i) = \emptyset$. An extended pattern $p_1 \setminus p_2$ is linear if p_1, p_2 are linear and $\mathcal{V}ar(p_1) \cap \mathcal{V}ar(p_2) = \emptyset$.

In what follows we consider that all (tuples of) extended patterns are linear and the set of all these patterns is denoted $\mathcal{T}_{\mathcal{E}}(\mathcal{C},\mathcal{X})$.

The instance relation can be extended to take into account extended patterns:

$$\begin{array}{rcl} p_1+p_2 & \prec\!\!\prec & v \quad \text{iff } p_1 \prec\!\!\prec v \vee p_2 \prec\!\!\prec v \\ p_1 \setminus p_2 & \prec\!\!\prec & v \quad \text{iff } p_1 \prec\!\!\prec v \wedge p_2 \not\prec\!\!\prec v \\ \perp & \not\prec\!\!\prec & v \end{array}$$

with p_1, p_2 extended patterns and v value.

The notion of ground semantics is also extended to take into account the new constructions:

$$\begin{array}{rcl} [\![p_1+p_2]\!] & = & [\![p_1]\!] \cup [\![p_2]\!] \\ [\![p_1 \setminus p_2]\!] & = & [\![p_1]\!] \setminus [\![p_2]\!] \\ [\![\perp]\!] & = & \emptyset \end{array}$$

All notions apply as expected to tuples of extended patterns. We generally use the term extended pattern to designate an extended pattern or a tuple of extended patterns.

Proposition 2.2 (Instance relation vs. ground semantics for extended patterns)**.** *Given an extended pattern p and a value v, $v \in [\![p]\!]$ iff $p \prec\!\!\prec v$.*

The disambiguation problem can be generalized to extended patterns: given a list of extended patterns $[p_1, \ldots, p_n]$, the disambiguation problem consists thus in finding sets of *constructor* patterns $P_1, \ldots, P_n$ such that for each $i \in [1..n]$, $[\![P_i]\!] = [\![p_i]\!] \setminus \cup_{j=1}^{i-1} [\![p_j]\!]$. When restricting to lists of constructor patterns we retrieve the original disambiguation problem. By abuse of language, when we refer to the disambiguation of a pattern we mean the disambiguation of the list consisting only of this pattern; when the pattern is constructor the disambiguation obviously results in the list containing only this pattern. Supposing this generalized disambiguation problem can be solved, the definition of a function by a list of equations involving extended patterns can be replaced by an equivalent one consisting of a set of equations using only constructor patterns.

3 Encoding extended patterns

To solve the disambiguation problem we propose a method for transforming any extended pattern p and, in particular, any complement pattern, into an equivalent pure additive pattern $p_1 + \cdots + p_n$ and thus obtain the set of constructor patterns $\{p_1, \cdots, p_n\}$ having the same semantics as the original one; if p is transformed into $\perp$ then it is useless. This transformation is accomplished using the rewriting system $\mathfrak{R}_\setminus$ presented in Figure 1. For simplicity, this rewriting system is presented schematically using rules which abstract over the symbols of the signature. We use overlined symbols, like $\bar{t}$, $\overline{v}, \overline{w}$, to denote the variables of the TRS and z to denote (freshly generated) pattern-level variables. We will show that each intermediate step and consequently the overall transformation is sound and complete *w.r.t.* the ground semantics.

Rules $A1$ and $A2$ express the fact that the empty ground semantics of $\perp$ is neutral for the union. Rule $E1$ indicates that the semantics of a pattern containing a sub-term with an empty ground semantics is itself empty. Similarly, if the semantics of a sub-term can be expressed as the union of two sets then the semantics of the overall term is obtained by distributing these sets over the corresponding constructors; this behaviour is reflected by the rule $S1$. Note that $E1$ and $S1$ are rule schemes representing as many rules as constructors of strictly positive arity in the signature and tuple symbols in $\mathcal{L}$.

The remaining rules describe the behaviour of complements and generally correspond to set theory laws over the ground semantics of the involved patterns. The difference between the ground semantics

Remove empty sets:

(A1)	$\bot + \overline{v}$	$\Rightarrow$	$\overline{v}$	
(A2)	$\overline{v} + \bot$	$\Rightarrow$	$\overline{v}$	

Distribute sets:

(E1)	$h(\overline{v_1},\ldots,\bot,\ldots,\overline{v_n})$	$\Rightarrow$	$\bot$	
(S1)	$h(\overline{v_1},\ldots,\overline{v_i}+\overline{w_i},\ldots,\overline{v_n})$	$\Rightarrow$	$h(\overline{v_1},\ldots,\overline{v_i},\ldots,\overline{v_n}) + h(\overline{v_1},\ldots,\overline{w_i},\ldots,\overline{v_n})$	

Simplify complements:

(M1)	$\overline{v} \setminus \overline{V}$	$\Rightarrow$	$\bot$	
(M2)	$\overline{v} \setminus \bot$	$\Rightarrow$	$\overline{v}$	
(M3)	$\overline{w} \setminus (\overline{v_1} + \overline{v_2})$	$\Rightarrow$	$(\overline{w} \setminus \overline{v_1}) \setminus \overline{v_2}$	
(M4)	$\overline{V} \setminus g(\overline{t_1},\ldots,\overline{t_n})$	$\Rightarrow$	$\sum_{c\in\mathcal{C}} c(z_1,\ldots,z_m) \setminus g(\overline{t_1},\ldots,\overline{t_n})$	*with* $m = arity(c)$
(M5)	$\bot \setminus f(\overline{v_1},\ldots,\overline{v_n})$	$\Rightarrow$	$\bot$	
(M6)	$(\overline{v}+\overline{w}) \setminus f(\overline{v_1},\ldots,\overline{v_n})$	$\Rightarrow$	$(\overline{v} \setminus f(\overline{v_1},\ldots,\overline{v_n})) + (\overline{w} \setminus f(\overline{v_1},\ldots,\overline{v_n}))$	
(M7)	$f(\overline{v_1},\ldots,\overline{v_n}) \setminus f(\overline{t_1},\ldots,\overline{t_n})$	$\Rightarrow$	$f(\overline{v_1} \setminus \overline{t_1},\ldots,\overline{v_n}) + \cdots + f(\overline{v_1},\ldots,\overline{v_n} \setminus \overline{t_n})$	
(M8)	$f(\overline{v_1},\ldots,\overline{v_n}) \setminus g(\overline{w_1},\ldots,\overline{w_n})$	$\Rightarrow$	$f(\overline{v_1},\ldots,\overline{v_n})$	*with* $f \neq g$

Figure 1: $\mathfrak{R}_\setminus$: reduce extended patterns to additive terms. $\overline{v},\overline{v_1},\ldots,\overline{v_n}$, $\overline{w},\overline{w_1},\ldots,\overline{w_n}$ range over additive patterns, $\overline{t_1},\ldots,\overline{t_n}$ range over pure additive patterns, $\overline{V}$ ranges over pattern variables. f,g expand to all the symbols in $\mathcal{C}\cup\mathcal{L}$, h expands to all symbols in $\mathcal{C}^{n>0}\cup\mathcal{L}$.

of any pattern and the ground semantics of a variable, which corresponds to the set of all ground constructor patterns for the signature, is the empty set; rule $M1$ encodes this behaviour. When subtracting the empty set, the argument remains unchanged (rule $M2$). Subtracting the union of several sets consists in subtracting successively all sets (rule $M3$). The semantics of a variable is the set of all ground constructor patterns, set which can be also obtained by considering for each constructor in the signature the set of all terms having this symbol at the root position and taking the union of all these sets (rule $M4$). We should emphasize that $\overline{V}$ is a variable ranging over pattern variables at the object level and that z_i are fresh pattern variables seen as constants at the TRS level (*i.e.* $\overline{V}$ matches any z_i). Similarly to rules $M1-M3$, rules $M5$ and $M6$ correspond to their counterparts from set theory. Rule $M7$ corresponds to the set difference of cartesian products; the case when the head symbol is a constant c corresponds to the rule $c \setminus c \Rightarrow \bot$. Rule $M8$ corresponds just to the special case where complemented sets are disjoint.

It is worth noticing that the rule schemes $M4-M8$ expand to all the possible rules obtained by replacing f,g with all the constructors in the original signature and all tuple symbols. Note also that the variables in the rewrite rules range over (pure) additive patterns which correspond implicitly to a call-by-vallue reduction strategy.

Example 3.1. *Let us consider the signature Σ with $\mathcal{C} = \{a,b,f\}$ and $ar(a) = ar(b) = 0$, $ar(f) = 2$. The pattern $f(x,y)\setminus f(z,a)$ corresponds to all patterns rooted by f but those of the form $f(z,a)$. According to rule $M7$ this corresponds to taking all patterns rooted by f which are not discarded by the first argument of $f(z,a)$, i.e. the pattern $f(x\setminus z,y)$, or by its second argument, i.e. the pattern $f(x,y\setminus a)$. We obtain thus the pattern $f(x\setminus z,y)+f(x,y\setminus a)$ which reduces, using rule $M1$ and the propagation and elimination of $\bot$ to $f(x,y\setminus a)$. Using rule $M4$ we obtain $f(x,(a+b+f(y_1,y_2))\setminus a)$ which reduces eventually to $f(x,b+f(y_1,y_2))$. We can then apply $S1$ to obtain the term $f(x,b)+f(x,f(y_1,y_2))$ which is irreducible.*

The rewrite rules apply also on tuples of patterns and $(\!|x,y|\!)\setminus(\!|z,a|\!)$ reduces using the same rules as above to $(\!|x,b|\!)+(\!|x,f(y_1,y_2)|\!)$. Similarly $(\!|x,y|\!)\setminus(\!|b,a|\!)$ reduces to $(\!|a+f(x_1,x_2),y|\!)+(\!|x,b+f(x_1,x_2)|\!)$ and *then to the irreducible term* $(\llparenthesis a,y\rrparenthesis + \llparenthesis f(x_1,x_2),y\rrparenthesis) + (\llparenthesis x,b\rrparenthesis + \llparenthesis x,f(x_1,x_2)\rrparenthesis)$.

Lemma 3.1 (Convergence)**.** *The rewriting system* $\mathfrak{R}_\setminus$ *is confluent and terminating. The normal form of an extended pattern w.r.t. to* $\mathfrak{R}_\setminus$ *is either* $\perp$ *or a sum of (tuples of) constructor patterns, i.e. a pure additive term* t *such that if* $t(\omega) = +$ *for a given* ω *then, for all* $\omega' < \omega$*,* $t(\omega') = +$*.*

Note that since the rewrite rules introduce only fresh pattern variables (rule $M4$) and duplicate terms only through $+$ (rules $M6$, $M7$ and $S1$), a linear term is always rewritten to a linear term and thus, the normal form of a linear term is linear as well.

As intuitively explained above, the reduction preserves the ground semantics of linear terms:

Proposition 3.2 (Complement semantics preservation)**.** *For any extended patterns* p, p'*, if* $p \Longrightarrow_{\mathfrak{R}_\setminus} p'$ *then* $[\![p]\!] = [\![p']\!]$*.*

Checking whether a (extended) pattern p is useless *w.r.t.* a set of patterns $\{p_1,\dots,p_n\}$ can be done by simply verifying that the pattern $p \setminus (p_1 + \cdots + p_n)$ is reduced by $\mathfrak{R}_\setminus$ to $\perp$, meaning that this pattern has an empty semantics:

Proposition 3.3 (Subsumption)**.** *Given the patterns* $p, p_1, \dots, p_n$*,* p *is subsumed by* $\{p_1,\dots,p_n\}$ *iff* $p \setminus (p_1 + \cdots + p_n) \downarrow_{\mathfrak{R}_\setminus} = \perp$*.*

Example 3.2. *We consider the signature in Example 3.1 and the list of patterns* $[\llparenthesis b,y\rrparenthesis, \llparenthesis a,b\rrparenthesis, \llparenthesis f(x,y),z\rrparenthesis, \llparenthesis x,b\rrparenthesis]$*. To check if the last pattern in the list is useless it is enough to verify whether the pattern* $\llparenthesis x,b\rrparenthesis \setminus (\llparenthesis b,y\rrparenthesis + \llparenthesis a,b\rrparenthesis + \llparenthesis f(x,y),z\rrparenthesis)$ *reduces to* $\perp$ *or not. The pattern* $\llparenthesis x,b\rrparenthesis \setminus \llparenthesis b,y\rrparenthesis$ *reduces to* $\llparenthesis a + f(x_1,x_2),b\rrparenthesis$ *and when we further subtract* $\llparenthesis a,b\rrparenthesis$ *we obtain* $\llparenthesis f(x_1,x_2),b\rrparenthesis$*. Finally,* $\llparenthesis f(x_1,x_2),b\rrparenthesis \setminus \llparenthesis f(x,y),z\rrparenthesis$ *reduces to* $\perp$ *and we can thus conclude that the pattern* $\llparenthesis x,b\rrparenthesis$ *is useless w.r.t. the previous patterns in the list.*

One may want to check the exhaustiveness of the list of patterns $[\llparenthesis b,y\rrparenthesis, \llparenthesis a,b\rrparenthesis, \llparenthesis f(x,y),z\rrparenthesis]$*. Since the pattern* $\llparenthesis x,y\rrparenthesis \setminus (\llparenthesis b,y\rrparenthesis + \llparenthesis a,b\rrparenthesis + \llparenthesis f(x,y),z\rrparenthesis)$ *reduces to* $\llparenthesis a,a\rrparenthesis$ *we can conclude that the property doesn't hold. We can then check similarly that exhaustiveness holds for the list of patterns* $[\llparenthesis b,y\rrparenthesis, \llparenthesis a,b\rrparenthesis, \llparenthesis f(x,y),z\rrparenthesis, \llparenthesis a,a\rrparenthesis]$*.*

With the transformation realized by $\mathfrak{R}_\setminus$ an extended pattern is transformed into an equivalent additive one with $\perp$ potentially present only at the root position and with all sums pushed at the top level (*i.e.* until there is no $+$ symbol below any other symbol). More precisely, if we abstract over the way $+$ associates, any extended pattern p is normalized *w.r.t.* $\mathfrak{R}_\setminus$ into $\perp$ or into a sum of (tuples of) constructor patterns $p_1 + \cdots + p_n$ having the same semantics as p. Since the semantics of this latter pattern is exactly the same as the semantics of the set $P = \{p_1,\dots,p_n\}$, the above transformations can be used to solve the disambiguation problem. If the result of the reduction of p is $\perp$ then $P = \emptyset$ and in this case the pattern is useless.

Example 3.3. *Let us consider the signature from Example 3.1 and the list of patterns* $[f(x,y), f(z,a)]$*. As we have seen, the pattern* $f(x,y) \setminus f(z,a)$ *reduces w.r.t.* $\mathfrak{R}_\setminus$ *to* $f(x,b) + f(x,f(y_1,y_2))$ *and thus, the original list of patterns is disambiguated into the sets of patterns* $\{f(x,y)\}$ *and* $\{f(x,b), f(x,f(y_1,y_2))\}$*.*

The above transformation can be also used as a generic compilation method for the so-called anti-terms [4], *i.e.* a method for transforming an anti-term into an extended pattern and eventually into a set of constructor patterns having the same semantics as the original anti-term. An anti-term is a linear term in $\mathcal{T}(\mathcal{C} \cup !, \mathcal{X})$[1] and, intuitively, the semantics of an anti-term represents the complement of its semantics with respect to $\mathcal{T}(\mathcal{C})$. Formally [4], $[\![t\,[!t']_\omega]\!] = [\![t\,[z]_\omega]\!] \setminus [\![t\,[t']_\omega]\!]$ where z is a fresh variable and for all $\omega' < \omega$, $t(\omega') \neq !$. For example, the complement of a variable $!x$ denotes $\mathcal{T}(\mathcal{C}) \setminus [\![x]\!] = \mathcal{T}(\mathcal{C}) \setminus \mathcal{T}(\mathcal{C}) = \emptyset$.

[1] In their most general form anti-terms are not necessarily linear.

Similarly, $!g(x)$ denotes $\mathcal{T}(\mathcal{C}) \setminus \{g(t) \mid t \in \mathcal{T}(\mathcal{C})\}$, and $f(!a,x)$ denotes $\{f(v,u) \mid v,u \in \mathcal{T}(\mathcal{C})\} \setminus \{f(a,u) \mid u \in \mathcal{T}(\mathcal{C})\}$.

The compilation is simply realized by replacing all anti-terms by their absolute complement; this replacement can be expressed by a single rewrite rule $\mathfrak{R}_! = \{!\bar{t} \Rightarrow z \setminus \bar{t}\}$ where $\bar{t}$ is a variable ranging over anti-terms and z corresponds to a *fresh* pattern-level variable (*i.e.* a variable of the pattern being transformed).

Example 3.4. *We have $f(x,!a) \Longrightarrow_{\mathfrak{R}_!} f(x,y \setminus a)$ where y is a fresh variable; this pattern reduces w.r.t. $\mathfrak{R}_\setminus$ to $f(x,b) + f(x,f(y_1,y_2))$. Similarly $!f(x,!a) \Longrightarrow_{\mathfrak{R}_!} z \setminus f(x,y \setminus a)$ with y,z fresh variables and the latter pattern reduces to $a+b+f(x,a)$.*

$\mathfrak{R}_!$ is clearly convergent and the normal form of any anti-term is an extended term containing no ! symbol. Since the reduction introduces only fresh variables and does not duplicate terms, the normal form of a linear term is linear as well. Moreover, the reduction preserves the ground semantics:

Proposition 3.4 (Anti-pattern semantics preservation)**.** *For any anti-terms $p,p' \in \mathcal{T}_\mathcal{E}(\mathcal{C} \cup \{!\}, \mathcal{X})$, if $p \Longrightarrow_{\mathfrak{R}_!} p'$ then, $[\![p]\!] = [\![p']\!]$.*

In the rest of this paper we will thus consider that an anti-pattern is just syntactic sugar for the corresponding extended pattern obtained by replacing all its sub-terms of the form $!q$ by $z \setminus q$ with z a fresh variable.

4 Elimination of redundant patterns

We have so far a method for transforming an extended (anti-)pattern p into a set of constructor patterns P. The set P is not necessarily canonical and can contain, for example, duplicate or redundant patterns, *i.e.* patterns useless *w.r.t.* the other patterns in P.

Example 4.1. *The pattern $f(x,!a) \setminus f(b,a)$ which corresponds to $f(x,y \setminus a) \setminus f(b,a)$ is reduced by $\mathfrak{R}_\setminus$ to $f(a+f(x_1,x_2),b+f(y_1,y_2)) + f(x,b+f(y_1,y_2))$ and finally to the pure additive pattern $f(a,b) + f(f(x_1,x_2),b) + f(a,f(y_1,y_2)) + f(f(x_1,x_2),f(y_1,y_2)) + f(x,b) + f(x,f(y_1,y_2))$. The disambiguation of the initial pattern results thus in the set $\{f(a,b),\ f(f(x_1,x_2),b),\ f(a,f(y_1,y_2)),\ f(f(x_1,x_2), f(y_1,y_2)),\ f(x,b),\ f(x,f(y_1,y_2))\}$ which is clearly equivalent to the set $\{f(x,b),\ f(x,f(y_1,y_2))\}$ since all the patterns of the former are subsumed by the patterns of the latter.*

The simplification consisting in eliminating patterns subsumed by other patterns is obvious and this is one of the optimizations proposed in Section 6. There are some other cases where a pattern is subsumed not by a single pattern but by several ones. The objective is to find, for each set P of constructor patterns resulting from the transformation of an extended pattern a smallest subset $P' \subseteq P$ such that P' has the same semantics as P. In particular, a pattern p_k from $P = \{p_1,\ldots,p_n\}$ can be removed from P without changing its semantics if $[\![p_k]\!] \subseteq \bigcup_{j \neq k} [\![p_j]\!]$. By exploring all possible removals we can find the smallest subset P.

Example 4.2. *We consider the signature from Example 3.1 enriched with the constructor g with $ar(g) = 1$ and the set of constructor patterns $\{f(g(b),f(x,b)), f(g(b),f(b,y)), f(g(x),f(a,b)), f(x,f(f(z_1,z_2),y)), f(x,f(g(z),y))\}$. This time none of the patterns is subsumed directly by another one but the first one is subsumed by the set consisting of the four other patterns. To convince ourselves we can consider instances of this pattern with x replaced respectively by $a,b,g(z)$ and $f(z_1,z_2)$ (i.e. all the constructors of the signature) and check that each of these instances is subsumed by one of the other patterns.*

$$
\begin{array}{l}
\mathit{minimum}(P) = \mathit{minimum}'(P,\emptyset) \\
\mathit{minimum}'(\emptyset,\mathit{kernel}) = \mathit{kernel} \\
\mathit{minimum}'(\{q\}\cup P,\mathit{kernel}) = \textbf{if } q \text{ is subsumed by } P\cup \mathit{kernel} \textbf{ then} \\
\qquad\qquad\qquad \mathit{smallest\_set}(\mathit{minimum}'(P,\{q\}\cup \mathit{kernel}),\mathit{minimum}'(P,\mathit{kernel})) \\
\qquad\qquad\quad \textbf{else} \\
\qquad\qquad\qquad \mathit{minimum}'(P,\{q\}\cup \mathit{kernel})
\end{array}
$$

Figure 2: $\mathit{minimum}(P)$ computes a minimal valid subset of P. $\mathit{smallest\_set}(P,P')$ returns P if $|P| < |P'|$, P' otherwise.

We have seen that we can identify redundant patterns in a set P (Proposition 3.3) and thus we can subsequently remove them in order to obtain a *valid subset* $P' \subseteq P$ with equivalent semantics, $[\![P']\!] = [\![P]\!]$. Given a set of patterns we can remove all redundant patterns one by one till the obtained set contains no such pattern but, depending on the pattern we have chosen to eliminate at some point, we can nevertheless get different valid subsets and some of them do not necessarily lead to a minimal one.

Computing the smallest valid subset can be done by enumerating the powerset of P and taking its smallest element P' which is a valid subset of P. Figure 2 presents a more efficient algorithm where the search space is reduced: P' is searched only among the subsets of P which contain the initial *kernel* of P, *i.e.* the set $\{p \mid p \in P, p \text{ is not subsumed by } P\setminus\{p\}\}$. The algorithm still explores all the possible valid subsets and eventually returns the minimal one:

Proposition 4.1 (Minimal subset)**.** *Given a set of constructor patterns P, the algorithm given in Figure 2 computes the smallest valid subset $P' \subseteq P$.*

5 Function encoding

We have focused so far on the matching mechanism behind function definitions using case expressions and we have eluded so far the potential problems related to the evaluation of such functions.

If we consider, for example, a function φ defined by the list of rules

$$
\begin{array}{l}
[\ \varphi(z,a) \twoheadrightarrow z, \\
\ \ \varphi(x,y) \twoheadrightarrow y\ \ ]
\end{array}
$$

we can proceed to the disambiguation of its patterns which results in the sets of patterns $\{(\!|z,a|\!)\}$ and $\{(\!|x,b|\!),(\!|x,f(y_1,y_2)|\!)\}$ as shown in the examples in the previous section. Consequently, if we replace naively the initial patterns with the ones obtained by disambiguation then the following set of corresponding rules is obtained

$$
\begin{array}{l}
\{\ \varphi(z,a) \twoheadrightarrow z, \\
\ \ \varphi(x,b) \twoheadrightarrow y, \\
\ \ \varphi(x,f(y_1,y_2)) \twoheadrightarrow y\ \ \}
\end{array}
$$

One can easily see that the two last rules are not well-defined and in what follows we extend the transformations proposed in the previous sections to tackle such situations.

Remove empty sets:

$$
\begin{array}{lrcll}
\text{(A1)} & \bot + \overline{v} & \Rightarrow & \overline{v} & \\
\text{(A2)} & \overline{v} + \bot & \Rightarrow & \overline{v} & \\
\end{array}
$$

Distribute sets:

$$
\begin{array}{lrcll}
\text{(E1)} & h(\overline{v_1},\ldots,\bot,\ldots,\overline{v_n}) & \Rightarrow & \bot & \\
\text{(E2)} & \overline{V} \,@\, \bot & \Rightarrow & \bot & \\
\text{(S1)} & h(\overline{v_1},\ldots,\overline{v_i}+\overline{w_i},\ldots,\overline{v_n}) & \Rightarrow & h(\overline{v_1},\ldots,\overline{v_i},\ldots,\overline{v_n}) + h(\overline{v_1},\ldots,\overline{w_i},\ldots,\overline{v_n}) & \\
\text{(S2)} & \overline{V} \,@\, (\overline{v_1}+\overline{v_2}) & \Rightarrow & \overline{V} \,@\, \overline{v_1} + \overline{V} \,@\, \overline{v_2} & \\
\end{array}
$$

Simplify complements:

$$
\begin{array}{lrcll}
\text{(M1)} & \overline{v} \setminus \overline{V} & \Rightarrow & \bot & \\
\text{(M2)} & \overline{v} \setminus \bot & \Rightarrow & \overline{v} & \\
\text{(M3)} & \overline{w} \setminus (\overline{v_1}+\overline{v_2}) & \Rightarrow & (\overline{w} \setminus \overline{v_1}) \setminus \overline{v_2} & \\
\text{(M4')} & \overline{V} \setminus g(\overline{t_1},\ldots,\overline{t_n}) & \Rightarrow & \overline{V} \,@\, (\sum_{c \in \mathcal{C}} c(z_1,\ldots,z_m) \setminus g(\overline{t_1},\ldots,\overline{t_n})) & \textit{with } m = arity(c) \\
\text{(M5)} & \bot \setminus f(\overline{v_1},\ldots,\overline{v_n}) & \Rightarrow & \bot & \\
\text{(M6)} & (\overline{v}+\overline{w}) \setminus f(\overline{v_1},\ldots,\overline{v_n}) & \Rightarrow & (\overline{v} \setminus f(\overline{v_1},\ldots,\overline{v_n})) + (\overline{w} \setminus f(\overline{v_1},\ldots,\overline{v_n})) & \\
\text{(M7)} & f(\overline{v_1},\ldots,\overline{v_n}) \setminus f(\overline{t_1},\ldots,\overline{t_n}) & \Rightarrow & f(\overline{v_1} \setminus \overline{t_1},\ldots,\overline{v_n}) + \cdots + f(\overline{v_1},\ldots,\overline{v_n} \setminus \overline{t_n}) & \\
\text{(M8)} & f(\overline{v_1},\ldots,\overline{v_n}) \setminus g(\overline{w_1},\ldots,\overline{w_n}) & \Rightarrow & f(\overline{v_1},\ldots,\overline{v_n}) & \textit{with } f \neq g \\
\text{(M9)} & \overline{V} \,@\, \overline{v} \setminus \overline{w} & \Rightarrow & \overline{V} \,@\, (\overline{v} \setminus \overline{w}) & \\
\text{(M10)} & \overline{v} \setminus \overline{V} \,@\, \overline{w} & \Rightarrow & \overline{v} \setminus \overline{w} & \\
\end{array}
$$

Figure 3: $\mathfrak{R}^{@}_{\setminus}$: reduce (as-)extended patterns to additive terms. $\overline{v},\overline{v_1},\ldots,\overline{v_n}$, $\overline{w},\overline{w_1},\ldots,\overline{w_n}$ range over additive patterns, $\overline{t_1},\ldots,\overline{t_n}$ range over pure additive patterns, $\overline{V}$ ranges over pattern variables. f,g expand to all the symbols in $\mathcal{C} \cup \mathcal{L}$, h expands to all symbols in $\mathcal{C}^{n>0} \cup \mathcal{L}$.

5.1 As-patterns and their encoding

We first consider a new construct for *extended patterns* which are now defined as follows:

$$p \quad := \quad \mathcal{X} \mid f(p_1,\ldots,p_n) \mid p_1 + p_2 \mid p_1 \setminus p_2 \mid \bot \mid q \,@\, p$$

with $f \in \mathcal{C}, q \in \mathcal{T}(\mathcal{C},\mathcal{X})$.

All patterns $q \,@\, p$, called *as-patterns*, are, as all the other extended patterns, linear *i.e.* p, q are linear and $\mathcal{V}ar(q) \cap \mathcal{V}ar(p) = \emptyset$. As we will see, @ is a convenient way to alias terms and use the variable name in the right-hand side of the corresponding rewrite rule. In fact, all the aliases used explicitly in the left-hand sides of the extended rules (defined formally in Section 5.3) are of the form $x \,@\, p$; the general form $q@p$ is used only in the matching process and in this case $q \in \mathcal{T}(\mathcal{F})$. @ has a higher priority than $\setminus$ which has a higher priority than $+$. From now on, unless stated explicitly, extended patterns are considered to include as-patterns.

The notion of ground semantics is extended accordingly for as-patterns: $[\![q \,@\, p]\!] = [\![q]\!] \cap [\![p]\!]$. Notice that the variable x aliasing the pattern p in $x \,@\, p$ has no impact on the semantics of the term: $[\![x \,@\, p]\!] = [\![p]\!]$.

To transform any extended (as-)pattern into a pure additive pattern we use the rewriting system $\mathfrak{R}^{@}_{\setminus}$ described in Figure 3; it consists of the the rules of $\mathfrak{R}_{\setminus}$ with the rule $M4$ slightly modified together with a set of specific rules used to handle the as-patterns.

The new rules $E2$ and $S2$ specify respectively that aliasing a $\perp$ is useless and that aliasing a sum comes to aliasing all its patterns. Rules $M9$ and $M10$ indicate that the alias of a complement pattern $p \setminus q$ concerns only the pattern p. The modified rule $M4'$ guarantees that the variables of a complement pattern are not lost in the transformation and, as we will see in the next sections, prevent ill-formed rules as those presented at the beginning of the section.

Example 5.1. *We consider the signature in Example 3.1. The pattern $f(x,y) \setminus f(z,a)$ reduces w.r.t. $\mathfrak{R}_{\setminus}^{@}$ as it had w.r.t. $\mathfrak{R}_{\setminus}$ but because of the new rule $M4'$ we obtain $f(x,y\,@\,(b+f(y_1,y_2)))$. This latter term is eventually reduced using rules $S2$ and $S1$ to $f(x,y\,@\,b)+f(x,y\,@\,f(y_1,y_2))$.*

$\mathfrak{R}_{\setminus}^{@}$ is convergent and the normal form of a term in $\mathcal{T}_{\mathcal{E}}(\mathcal{C},\mathcal{X})$ is similar to that that obtained with $\mathfrak{R}_{\setminus}$ but with some of its subterms potentially aliased with the @ construct.

Lemma 5.1 (Convergence)**.** *The rewriting system $\mathfrak{R}_{\setminus}^{@}$ is confluent and terminating. Given an extended pattern t the normal form of t w.r.t. to $\mathfrak{R}_{\setminus}^{@}$ is either $\perp$ or a sum of (tuples of) constructor patterns potentially aliased, i.e. a pure additive term t such that if $t(\omega)=+$ for a given ω then, for all $\omega' < \omega$, $t(\omega')=+$.*

For the same reasons as before, a linear term is always rewritten to a linear term and thus, the normal form of a linear term is linear as well. Once again, each intermediate step and consequently the overall transformation is sound and complete *w.r.t.* the ground semantics.

Proposition 5.2 (Complement semantics preservation)**.** *For any extended pattern p,p' if $p \Longrightarrow_{\mathfrak{R}_{\setminus}^{@}} p'$ then, $[\![p]\!] = [\![p']\!]$.*

5.2 Matchable and free variables

Given a constructor pattern p and a value v, if $v \in [\![p]\!]$ then there exists a substitution σ with $\mathcal{D}om(\sigma) = \mathcal{V}ar(p)$ *s.t.* $v \in [\![\sigma(p)]\!]$, or equivalently $v = [\![\sigma(p)]\!]$. When p is an extended pattern some of its variables are not significant for the matching, *i.e.* if $v \in [\![p]\!]$ then there exists a substitution σ *s.t.* $v \in [\![\tau(\sigma(p))]\!]$ for all substitution τ with $\mathcal{D}om(\tau) = \mathcal{V}ar(p) \setminus \mathcal{D}om(\sigma)$. For example, given the pattern $f(x, z \setminus g(y))$, the value $f(a,b)$ belongs to the semantics of any instance of $f(a, b \setminus g(y))$.

The set $\mathcal{MV}ar(p)$ of *matchable variables* of a pattern p is defined as follows:

$$
\begin{aligned}
&\mathcal{MV}ar(x) = \{x\}, \forall x \in \mathcal{X}\\
&\mathcal{MV}ar(f(p_1,\dots,p_n)) = \mathcal{MV}ar(p_1) \cup \ldots \cup \mathcal{MV}ar(p_n), \forall f \in \mathcal{C}^n\\
&\mathcal{MV}ar(p_1+p_2) = \mathcal{MV}ar(p_1) \cap \mathcal{MV}ar(p_2)\\
&\mathcal{MV}ar(p_1 \setminus p_2) = \mathcal{MV}ar(p_1)\\
&\mathcal{MV}ar(\perp) = \mathcal{X}\\
&\mathcal{MV}ar(q\,@\,p) = \mathcal{MV}ar(q) \cup \mathcal{MV}ar(p)
\end{aligned}
$$

The variables of p which are not matchable are *free*:

$$\mathcal{FV}ar(p) = \mathcal{V}ar(p) \setminus \mathcal{MV}ar(p)$$

Note that the definition of linearity we have used for (complement) patterns guarantees that matchable and free variables have different names. Consequently, we have $\sigma(p_1 \setminus p_2) = \sigma(p_1) \setminus p_2$ for all σ such that $\mathcal{D}om(\sigma) = \mathcal{MV}ar(p_1 \setminus p_2)$.

The encoding rules preserve not only the semantics but also the set of matchable variables of the initial pattern; this is important when transforming the extended rules introduced in the next section.

Proposition 5.3 (Variables preservation)**.** *For any extended patterns p, p' such that $p \Longrightarrow_{\mathfrak{R}_\backslash^@} p'$ we have $\mathcal{MV}ar(p) \subseteq \mathcal{MV}ar(p')$ and $\mathcal{FV}ar(p') \subseteq \mathcal{FV}ar(p)$.*

We consider for convenience that the set of matchable variables of $\perp$ is the set of all variables; a more natural definition considering the empty set would have required a more complicated statement for the above proposition dealing explicitly with the rules whose right-hand side is $\perp$. As explained in the next section this choice has no impact on the proposed formalism.

The above property was not verified by the rule $M4$ of $\mathfrak{R}_\backslash$ and it was intuitively the origin of the ill-formed rules presented at the beginning of the section. An immediate consequence of this property is that for any patterns $p, p_1, p_2, \ldots, p_n$ such that $p \downarrow_{\mathfrak{R}_\backslash^@} = p_1 + p_2 + \ldots + p_n$ we have $\mathcal{MV}ar(p) \subseteq \mathcal{MV}ar(p_i)$ for all $i \in [1, \ldots, n]$.

5.3 Encoding sets and lists of extended rules

An *extended rewrite rule*, or simply *extended rule*, over a signature Σ is a pair (l, r) (also denoted $l \twoheadrightarrow r$) with $l = \varphi(\overrightarrow{p})$, $\varphi \in \mathcal{D}$, $\overrightarrow{p}$ a tuple of extended patterns, and $r \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, such that $\mathcal{V}ar(r) \subseteq \mathcal{MV}ar(\overrightarrow{p})$. A set of extended rules $\mathcal{E} = \{\varphi(\overrightarrow{p_1}) \twoheadrightarrow r_1, \ldots, \varphi(\overrightarrow{p_n}) \twoheadrightarrow r_n\}$ induces a relation $\longrightarrow_{\mathcal{E}}$ over $\mathcal{T}(\mathcal{F})$ such that $t \longrightarrow_{\mathcal{E}} t'$ iff there exist $i \in [1, \ldots, n]$, $\omega \in \mathcal{P}os(t)$ and a substitution σ such that $t_{|\omega} = \varphi(\overrightarrow{v})$, $\mathcal{D}om(\sigma) = \mathcal{MV}ar(\overrightarrow{p_i})$, $\overrightarrow{v} \in [\![\sigma(\overrightarrow{p_i})]\!]$ and $t' = t\,[\sigma(r_i)]_\omega$. A list of extended rules $\mathcal{L} = [\varphi(\overrightarrow{p_1}) \twoheadrightarrow r_1, \ldots, \varphi(\overrightarrow{p_n}) \twoheadrightarrow r_n]$ induces a relation $\longrightarrow_{\mathcal{L}}$ over $\mathcal{T}(\mathcal{F})$ such that $t \longrightarrow_{\mathcal{L}} t'$ iff there exist $i \in [1, \ldots, n]$, $\omega \in \mathcal{P}os(t)$ and a substitution σ such that $t_{|\omega} = \varphi(\overrightarrow{v})$, $\mathcal{D}om(\sigma) = \mathcal{MV}ar(\overrightarrow{p_i})$, $\overrightarrow{v} \in [\![\sigma(\overrightarrow{p_i})]\!]$, $p_j \not\prec\!\!\prec v, \forall j < i$ and $t' = t\,[\sigma(r_i)]_\omega$. We may write $\varphi(\overrightarrow{v}) \xrightarrow{i}_{\mathcal{L}} t'$ to indicate that the i-th extended rule has been used in the reduction.

When restricting to constructor patterns, all variables of a pattern p are matchable and thus, for all σ such that $\mathcal{D}om(\sigma) = \mathcal{MV}ar(p)$, $\sigma(p)$ is ground and for all value v we have $v \in [\![\sigma(p)]\!]$ iff $v = \sigma(p)$. Consequently, sets and lists of extended rules whose left-hand sides contain only constructor patterns are nothing else but CTRSs and respectively ordered CTRSs. We have used a different syntax for extended rules and general rules in order to help the reader identify the rules used in function definitions from those used to transform the extended patterns.

In the rest of this section we show how these lists of extended rules, which can be seen as extended ordered CTRSs, can be compiled towards plain ordered CTRS and eventually plain (order independent) CTRS.

Note that since we considered that the set of matchable variables of $\perp$ is the set of all variables, we can have extended rules with a left-hand side $\perp$ and an arbitrary right-hand. Such extended rules are of no practical use since they apply on no term. For the rest of this paper we restrict thus to extended rules containing no $\perp$.

Example 5.2. *We consider a signature with $\mathcal{D} = \{\varphi\}$, $\mathcal{C} = \{a, b, f\}$, $ar(\varphi) = 2$, $ar(a) = ar(b) = 0$, $ar(f) = 2$ and the list of extended rules $\mathcal{L} = [\varphi(x, y\,@\,!a) \twoheadrightarrow y, \varphi(a+b, y) \twoheadrightarrow y, \varphi(f(x,y), z) \twoheadrightarrow x]$. We have the reductions $\varphi(a,b) \xrightarrow{1}_{\mathcal{L}} b$, $\varphi(a,a) \xrightarrow{2}_{\mathcal{L}} a$, $\varphi(b,a) \xrightarrow{2}_{\mathcal{L}} a$, $\varphi(f(b,a),a) \xrightarrow{3}_{\mathcal{L}} b$. We also have $\varphi(\varphi(a,a), \varphi(f(b,a),a)) \xrightarrow{3}_{\mathcal{L}} \varphi(\varphi(a,a), b) \xrightarrow{2}_{\mathcal{L}} \varphi(a,b) \xrightarrow{1}_{\mathcal{L}} b$.*

The semantics preservation guaranteed by $\mathfrak{R}_\backslash^@$ has several consequences. On one hand, the corresponding transformations can be used to check the completeness of an equational definition and on the other hand, they can be used to transform extended patterns into equivalent constructor patterns (potentially aliased) and eventually to compile prioritized equational definitions featuring extended patterns into classical and order independent definitions. The former was detailed in Section 4. For the latter, we proceed in several steps and we define first a transformation $\mathfrak{T}^{\backslash}$ which encodes a list of extended rules $\mathcal{L} = [\varphi(\overrightarrow{p_1}) \twoheadrightarrow t_1, \ldots, \varphi(\overrightarrow{p_n}) \twoheadrightarrow t_n]$ into a list of rules using only constructor patterns potentially aliased:

$$\begin{aligned}\mathfrak{T}^{\backslash}(\mathcal{L}) = \oplus_{i=1}^{n}[\ & \varphi(\overrightarrow{q_1^i}) \twoheadrightarrow t_i, \ldots, \varphi(\overrightarrow{q_m^i}) \twoheadrightarrow t_i \mid \\ & \overrightarrow{q_1^i} + \ldots + \overrightarrow{q_m^i} = \overrightarrow{p_i} \downarrow_{\mathfrak{R}^{@}_{\backslash}} \neq \perp, \\ & \overrightarrow{q_1^i}, \ldots, \overrightarrow{q_m^i} \text{ contain no symbol } +]\end{aligned}$$

The order should be preserved in the resulting list of rules, *i.e.* the rules obtained for a given extended rule should be placed in the resulting list at the corresponding position. The transformation can be applied in a similar way to sets of extended rules.

Note that if the left-hand side of a rule reduces to $\perp$ then, there is no corresponding rule in the result of the transformation. According to Proposition 5.3, $\mathcal{MVar}(p_i) \subseteq \mathcal{MVar}\left(q_j^i\right)$ for all $j \in [1, \ldots, m]$ and since $\mathfrak{R}^{@}_{\backslash}$ preserves the linearity then, the rules in $\mathfrak{T}^{\backslash}(\mathcal{L})$ are all well-formed.

Example 5.3. *We consider the signature and the list of extended rules in Example 5.2. The pattern $\varphi(x, y @ !a)$ of the first extended rule reduces w.r.t. $\mathfrak{R}^{@}_{\backslash}$ to $f(x, y @ b) + f(x, y @ f(y_1, y_2))$ (reductions are similar to those in Example 3.4 but we also consider aliasing now). The pattern $\varphi(a+b, y)$ reduces immediately to the pattern $\varphi(a, y) + \varphi(b, y)$. We have thus $\mathfrak{T}^{\backslash}(\mathcal{L}) = [f(x, y @ b) \twoheadrightarrow y, f(x, y @ f(y_1, y_2)) \twoheadrightarrow y$, $\varphi(a, y) \twoheadrightarrow y, \varphi(b, y) \twoheadrightarrow y, \varphi(f(x, y), z) \twoheadrightarrow x]$.*

We can then check that the reductions presented in Example 5.2 are still possible with $\mathfrak{T}^{\backslash}(\mathcal{L})$: $\varphi(a, b) \xrightarrow{1}_{\mathfrak{T}^{\backslash}(\mathcal{L})} b$, $\varphi(a, a) \xrightarrow{3}_{\mathfrak{T}^{\backslash}(\mathcal{L})} a$, $\varphi(b, a) \xrightarrow{4}_{\mathfrak{T}^{\backslash}(\mathcal{L})} a$ and $\varphi(f(b, a), a) \xrightarrow{5}_{\mathfrak{T}^{\backslash}(\mathcal{L})} b$. As before, we have also $\varphi(\varphi(a, a), \varphi(f(b, a), a)) \xrightarrow{5}_{\mathfrak{T}^{\backslash}(\mathcal{L})} \varphi(\varphi(a, a), b) \xrightarrow{3}_{\mathfrak{T}^{\backslash}(\mathcal{L})} \varphi(a, b) \xrightarrow{1}_{\mathcal{L}} b$.

The transformation preserves the corresponding relations:

Proposition 5.4 (Complement encoding)**.** *Given a list of extended rules $\mathcal{L}$ and a term $t \in \mathcal{T}(\mathcal{F})$, we have $t \longrightarrow_{\mathcal{L}} t'$ iff $t \longrightarrow_{\mathfrak{T}^{\backslash}(\mathcal{L})} t'$.*

The left-hand sides of the rules obtained following the $\mathfrak{T}^{\backslash}$ transformation are constructor patterns potentially aliased and to remove the aliases from these rules and replace accordingly the concerned variables in the corresponding right-hand sides we use the following recursive transformation:

$$\begin{aligned}\mathfrak{T}^{@}(\varphi(p) \twoheadrightarrow r) &= \varphi(p) \twoheadrightarrow r \\ & \quad \text{if } \forall \omega \in \mathcal{P}os(p), p(\omega) \neq @ \\ \mathfrak{T}^{@}(\varphi(p[x @ q]_{\omega}) \twoheadrightarrow r) &= \mathfrak{T}^{@}(\varphi(p[q]_{\omega}) \twoheadrightarrow \{x \mapsto q\}r) \\ & \quad \text{if } \forall \omega' \in \mathcal{P}os(q), q(\omega') \neq @\end{aligned}$$

which extends to lists ans sets of extended rules:

$$\mathfrak{T}^{@}([e_1, \ldots, e_n]) = [\mathfrak{T}^{@}(e_1), \ldots, \mathfrak{T}^{@}(e_n)]$$

$$\mathfrak{T}^{@}(\{e_1, \ldots, e_n\}) = \{\mathfrak{T}^{@}(e_1), \ldots, \mathfrak{T}^{@}(e_n)\}$$

Note that since at each intermediate transformation step the considered aliased pattern q contains no aliases itself then the right-hand sides $\{x \mapsto q\}r$ of the obtained rules contain no aliases at the positions concerned by the replacement and thus, become eventually terms in $\mathcal{T}(\mathcal{F}, \mathcal{X})$.

Example 5.4. *We consider the list of rules obtained by applying the transformation $\mathfrak{T}^{\backslash}$ in Example 5.3: $\mathfrak{T}^{\backslash}(\mathcal{L}) = [f(x, y @ b) \twoheadrightarrow y, f(x, y @ f(y_1, y_2)) \twoheadrightarrow y, \varphi(a, y) \twoheadrightarrow y, \varphi(b, y) \twoheadrightarrow y, \varphi(f(x, y), z) \twoheadrightarrow x]$. We have then $\mathfrak{T}^{@}(\mathfrak{T}^{\backslash}(\mathcal{L})) = [f(x, b) \twoheadrightarrow b, f(x, f(y_1, y_2)) \twoheadrightarrow f(y_1, y_2), \varphi(a, y) \twoheadrightarrow y, \varphi(b, y) \twoheadrightarrow y, \varphi(f(x, y), z) \twoheadrightarrow x]$.*

The transformation $\mathfrak{T}^{@}$ obviously preserves the ground semantics of the left-hand side of the transformed rules and terminates since it decreases the number of aliases in the left-hand side. The result of the transformation preserves the one-step semantics of the original rules:

Proposition 5.5 (Alias encoding)**.** *Given a list of rules $\mathcal{L}$ using only constructor as-patterns and a term $t \in \mathcal{T}(\mathcal{F})$, we have $t \longrightarrow_{\mathcal{L}} t'$ iff $t \longrightarrow_{\mathfrak{T}^{@}(\mathcal{L})} t'$.*

We have thus a method allowing the transformation of a list of rules using extended patterns, *i.e.* complement patterns (and thus anti-patterns), sum patterns and as-patterns, into an equivalent list using only constructor patterns. This is particularly useful when we have to compile towards languages which can handle only this kind of patterns. We also have the ingredients to encode it into a set of rules; this is interesting when we want to use (reasoning) tools for which is more convenient to handle order independent rules. For this, we replace the pattern of each rule in a list by complementing it *w.r.t.* the previous rules in the list. More precisely, given a list of rules $\mathcal{L} = [\varphi(\overrightarrow{p_1}) \twoheadrightarrow t_1, \ldots, \varphi(\overrightarrow{p_n}) \twoheadrightarrow t_n]$ we consider the transformation

$$\begin{array}{rl}\mathfrak{T}^{<}(\mathcal{L}) = \cup_{i=1}^{n}\{ & \varphi(\overrightarrow{q_1^i}) \twoheadrightarrow t_i, \ldots, \varphi(\overrightarrow{q_m^i}) \twoheadrightarrow t_i \mid \\ & \overrightarrow{q_1^i} + \ldots + \overrightarrow{q_m^i} = \overrightarrow{p_i} \setminus (\overrightarrow{p_1} + \ldots + \overrightarrow{p_{i-1}}) \downarrow_{\mathfrak{R}^{@}_{\setminus}} \neq \bot, \\ & \overrightarrow{q_1^i}, \ldots, \overrightarrow{q_m^i} \text{ contain no symbol } + \}\end{array}$$

which preserves the initial relation:

Proposition 5.6 (Order encoding)**.** *Given a list of extended rule $\mathcal{L}$ and a term $t \in \mathcal{T}(\mathcal{F})$, we have $t \longrightarrow_{\mathcal{L}} t'$ iff $t \longrightarrow_{\mathfrak{T}^{<}(\mathcal{L})} t'$.*

Example 5.5. *We consider the signature in Example 5.3 and the list of rules $\mathcal{L} = [\varphi(z,a) \twoheadrightarrow z, \varphi(x,y) \twoheadrightarrow y]$. The pattern $(\!|x,y|\!) \setminus (\!|z,a|\!)$ reduces w.r.t. $\mathfrak{R}^{@}_{\setminus}$ to $f(x, y \,@\, b) + f(x, y \,@\, f(y_1,y_2))$ (reductions are similar to those in Example 3.1 and Example 3.3 but we also consider aliasing now). We have thus $\mathfrak{T}^{<}(\mathcal{L}) = \{\varphi(z,a) \twoheadrightarrow z, \varphi(x, y \,@\, b) \twoheadrightarrow y, \varphi(x, y \,@\, f(y_1,y_2)) \twoheadrightarrow y\}$ and $\mathfrak{T}^{@}(\mathfrak{T}^{<}(\mathcal{L})) = \{\varphi(z,a) \twoheadrightarrow z,\ \varphi(x,b) \twoheadrightarrow b,$ $\varphi(x, f(y_1,y_2)) \twoheadrightarrow f(y_1,y_2)\}$.*

The transformations presented in this section preserve the one step semantics of the transformed lists (or sets) of rules. Moreover, the transformations are well-defined, *i.e.* produce lists or sets of well-formed rules. The produced rules have a specific shape suitable for subsequent transformations. We can thus combine them to define a transformation $\mathfrak{T}^{\mathfrak{ap}} = \mathfrak{T}^{@} \circ \mathfrak{T}^{\setminus}$ which transforms a list of rules using extended patterns into one using only constructor patterns, or a transformation $\mathfrak{T} = \mathfrak{T}^{@} \circ \mathfrak{T}^{<}$ which transforms a list of rules (using extended patterns) into a set of rules using only plain constructor patterns, *i.e.* a CTRS.

Corollary 1 (Simulation)**.** *Given a list of extended rules $\mathcal{L}$ and a term $t \in \mathcal{T}(\mathcal{F})$,*

1. *$t \longrightarrow_{\mathcal{L}} t'$ iff $t \longrightarrow_{\mathfrak{T}^{\mathfrak{ap}}(\mathcal{L})} t'$;*
2. *$t \longrightarrow_{\mathcal{L}} t'$ iff $t \longrightarrow_{\mathfrak{T}(\mathcal{L})} t'$.*

6 Optimizations and experimental results

We know that our algorithms can exhibit exponential time behavior since the useful clause problem is NP-complete [17]. Since we target practical implementations with reasonable running times, we identify the critical aspects leading to this exponential behavior and try to limit as much as possible this explosion. The exponential behavior originates in our case from the rules $M4$ and $M7$ (Figure 1) for which the number of elements in the generated sums determines the effective branching. We propose two optimizations that turn out to limit significantly this number for concrete examples.

Cut useless choices. For a symbol f of arity $n > 0$, the rule M7 transforms the term $f(\overline{v_1},\ldots,\overline{v_n}) \setminus f(\overline{t_1},\ldots,\overline{t_n})$ into a sum $\sum_{i=1}^{n} f(\overline{v_1},\ldots,\overline{v_i} \setminus \overline{t_i},\ldots,\overline{v_n})$ of n new terms to reduce. We can remark that if there exists a k such that $\overline{v_k} \setminus \overline{t_k} = \overline{v_k}$ then the k-th term of the sum is the term $f(\overline{v_1},\ldots,\overline{v_n})$ which subsumes all the other terms in the sum and whose semantics is thus the same as that of the sum. Therefore, as soon as such a term is exhibited the sum can be immediately reduced to $f(\overline{v_1},\ldots,\overline{v_n})$ avoiding thus further unnecessary reductions. For example, the term $f(x,a) \setminus f(a,b)$ normally reduces to $f(x \setminus a,a) + f(x,a \setminus b)$ which is eventually reduced to $f(x,a)$, while using the optimization we get directly $f(x,a)$.

Sorted encoding. Given a term of the form $\overline{V} \setminus g(\overline{t_1},\ldots,\overline{t_n})$, the rule M4 produces a sum $\sum_{f\in\mathcal{C}} f(z_1,\ldots,z_m) \setminus g(\overline{t_1},\ldots,\overline{t_n})$ containing an element for each constructor f of the signature. In practice, algebraic signatures are often many-sorted and in this case, since we can always identify the sort of the variable $\overline{V}$ in a given context then, the sum $\sum_{f\in\mathcal{F}} f(z_1,\ldots,z_m)$ can be restricted to all the constructors of this sort.

Example 6.1. *Let us consider the many-sorted signature $E = a \mid b \mid c$ and $L = cons(E,L) \mid nil$. The application of rule M4 to the term $cons(z \setminus a, nil)$ produces $cons(z\,@\,((a+b+c+nil+cons(z_1,z_2)) \setminus a), nil)$ which is reduced, by propagation of $\setminus$, to $cons(z\,@\,(a \setminus a + b \setminus a + c \setminus a + nil \setminus a + cons(z_1,z_2) \setminus a), nil)$. This term contains ill-typed terms like $nil \setminus a$ or $cons(z_1,z_2) \setminus a$ that are eventually reduced to $\perp$ and eliminated. With the optimization, we infer the type E for z and generate directly the correctly typed term $cons(z\,@\,((a+b+c) \setminus a), nil)$.*

Improved minimization. The minimization algorithm has also an exponential complexity on the number n of input rules. A first optimization follows the observation that if a pattern l_i subsumes a pattern l_j then, l_j cannot be in the minimal set of patterns. We can thus safely eliminate all patterns directly subsumed by another one right from the beginning and decrease the number of recursive calls accordingly.

A second optimization consists in initializing the kernel with all the patterns l_i which are not subsumed by $l_1,\ldots,l_{i-1},l_{i+1},\ldots,l_n$. Indeed, for such l_i the test "l_i is subsumed by $S \cup kernel$" is always false and thus, l_i is added systematically to the kernel during the computation. Initializing the kernel with these l_i reduces the complexity from $O(2^n)$ to $O(2^{n-k})$, where k is the number of such patterns.

Local minimum vs. global minimum. Given a set of terms S, the algorithm given in Figure 2 computes the smallest subset $S' \subseteq S$ such that S' is a valid subset of S. In the general case S' may not be the smallest set such that $[\![S']\!] = [\![S]\!]$ and we show how our algorithm can be used to find this smallest set.

We consider the *saturation* $\overline{S}$ of a set of terms S *w.r.t.* to all terms subsuming terms in S and preserving the semantics: $\overline{S} = \cup_{q\in S}\{p \mid [\![q]\!] \subseteq [\![p]\!] \subseteq [\![S]\!]\}$. We can show that the minimum of the saturated set is smaller than the minimum of the original one, $|minimum(\overline{S})| \leq |minimum(S)|$, and that the minimum of the saturations of two sets of patterns with the same semantics $[\![S]\!] = [\![S']\!]$ is the same, $minimum(\overline{S}) = minimum(\overline{S'})$. We can thus, take any of them and compute its saturation and the corresponding (local) minimal set of patterns. Since the minimum of a saturated set is smaller than any of its subsets then, the obtained minimum is global:

Proposition 6.1. *$minimum(\overline{S})$ is the (global) minimum valid subset of S.*

This is not an actual optimization but just an extension which guarantees the global minimality. For most of the examples we have experienced with, this global minimization technique had no impact, the minimum set of patterns being obtained directly by our rule elimination.

Implementation. All the transformations and optimizations presented in the paper have been implemented in Tom, an extension of Java allowing the use of rules and strategies. This implementation[2] can generate TRSs expressed in several syntaxes like, for example, AProVE[7]/TTT2[11] syntax which can be used to check termination, and Tom syntax which can be used to execute the resulting TRS.

[2] `https://github.com/rewriting/tom`, `[scm]/applications/strategyAnalyzer/`

An alternative Haskell implementation[3] has allowed us to generate via GHCJS a javascipt version of the algorithm that can be experimented in a browser [4].

For simplicity, we presented the formalism in a mono-sorted context but both implementations handle many-sorted signatures and implement the corresponding optimization explained above.

7 Related works

A. Krauss studied the problem of transforming function definitions with pattern matching into minimal sets of independent equations [12]. Our approach can be seen as a new method for solving the same problem but arguably easier to implement because of the clear and concise rule based specification. We handle here the right-hand sides of the rewrite rules and allow anti-patterns in the left-hand sides of rules and although this seems also feasible with the method in [12] the way it can be done is not made explicit in the paper. We couldn't obtain the prototype mentioned in the paper but when experimenting with the proposed examples we obtained execution times which indicate comparable performances to those given in [12] and, more importantly, identical results:

- Interp is an interpreter for an expression language defined by 7 ordered rules for which a naive disambiguation would produce 36 rules [12]. Our transformation without minimization produces 31 rules and using the minimization algorithm presented in Section 4 we obtain, as in [12], 25 rules. Example 4.2 is indeed inspired by one of the rules eliminated during the minimization process for this specification.
- Balance is a balancing function for red-black-trees. A. Krauss reported that for this list of 5 ordered rules a naive approach would produce 91 rules. In our case, the transformation directly produces the minimal set composed of 59 rules reported in [12].
- Numadd is a function that operates on arithmetic expressions. It is composed of 5 rules. Our transformation directly produces the minimal set composed of 256 rules [12].

L. Maranget has proposed an algorithm for detecting useless patterns [14] for OCaml and Haskell. As mentioned previously, the algorithm in Figure 1 can be also used to check whether a pattern is useless *w.r.t.* a set of patterns (Proposition 3.3) but it computes the difference between the pattern and the set in order to make the decision. The minimization algorithm in Figure 2 can thus use the two algorithms interchangeably. Both algorithms have been implemented and we measured the execution time for the minimization function on various examples. In average, L. Maranget's approach is 30% more efficient than ours (40ms vs 55ms for the Interp example for instance) and can thus be used in the minimization algorithm if one wants to gain some efficiency with the price of adding an auxiliary algorithm.

This work has been initially motivated by our encoding of TRSs guided by rewriting strategies into plain TRSs [5]. This encoding produces intermediate systems of the form $\{\varphi(l) \twoheadrightarrow r, \varphi(x \,@\, !l) \twoheadrightarrow r', \ldots\}$ which are eventually reduced by expanding the anti-patterns into plain TRSs. Alternatively, we could use a simpler compilation schema based on ordered CTRSs in which case the intermediate system would have the form $[\varphi(l) \twoheadrightarrow r, \varphi(x) \twoheadrightarrow r', \ldots]$ and then, apply the approach presented in this paper to transform the resulting ordered CTRS into an order independent CTRS. We experimented with this new approach and for all the examples in [5] we obtained between 20% and 25% less rules than before. When using strategies, the order of rule application is expressed with a left-to-right strategy choice-operator and for

[3] http://github.com/polux/subsume

[4] http://htmlpreview.github.io/?https://github.com/polux/subsume/blob/web/out/index.html

such strategies the gain with our new approach is even more significant than for the examples in [5] which involved only plain, order independent, TRSs.

There are a lot of works [9, 13, 1, 8] targeting the analysis of functional languages essentially in terms of termination and complexity, and usually they involve some encoding of the match construction. These encodings are generally deep and take into account the evaluation strategy of the targeted language leading to powerful analyzing tools. Our encodings are shallow and independent of the reduction strategy. Even if it turned out to be very practical for encoding ordered CTRSs involving anti-patterns and prove the (innermost) termination of the corresponding CTRSs with AProVE/TTT2, in the context of functional program analysis we see our approach more like a helper that will be hopefully used as an add-on by the existing analyzing tools.

8 Conclusion

We have proposed a concise and clear algorithm for computing the complement of a pattern *w.r.t.* a set of patterns and we showed how it can be used to encode an OTRS potentially containing anti-patterns into a plain TRS preserving the one-step semantics of the original system. The approach can be used as a generic compiler for ordered rewrite systems involving anti-patterns and, in collaboration with well-established techniques for TRS, for analyzing properties of such systems. Since the TRSs obtained with our method define exactly the same relation over terms, the properties of the TRS stand also for the original OTRS and the counter-examples provided by the analyzing tools when the property is not valid can be replayed directly for the OTRS. Moreover, our approach can be used as a new method for detecting useless patterns and for the minimization of sets of patterns.

For all the transformations we have performed the global minimization technique was superfluous and we conjecture that, because of the shape of the problems we handle and of the way they are handled, the transformation using a local minimization produces directly the smallest TRS for any input OTRS. One of our objectives is to prove this conjecture.

We consider of course integrating our algorithm into automatic tools either to disambiguate equational specifications or to (dis)prove properties of such specifications. The two available implementations let us think that such an integration can be done smoothly for any tool relying on a declarative programming language.

Acknowledgments. We want to thank Paul Brauner who implemented the Haskell version of the algorithm and who gave us very helpful feedback on previous drafts of this paper.

References

[1] M. Avanzini, U. D. Lago, and G. Moser. Analysing the complexity of functional programs: higher-order meets first-order. In *ICFP 2015*, pages 152–164, 2015.

[2] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[3] E. Balland, P. Brauner, R. Kopetz, P.-E. Moreau, and A. Reilles. Tom: Piggybacking rewriting on java. In *RTA '07*, volume 4533 of *LNCS*, pages 36–47. Springer-Verlag, 2007.

[4] H. Cirstea, C. Kirchner, R. Kopetz, and P.-E. Moreau. Anti-patterns for rule-based languages. *J. of Symbolic Computation*, 45(5):523 – 550, 2010.

[5] H. Cirstea, S. Lenglet, and P. Moreau. A faithful encoding of programmable strategies into term rewriting systems. In *RTA 2015*, volume 36 of *LIPIcs*, pages 74–88, 2015.

[6] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. The Maude 2.0 System. In *RTA '03*, volume 2706 of *LNCS*, pages 76–87. Springer-Verlag, 2003.

[7] C. Fuhs, J. Giesl, M. Parting, P. Schneider-Kamp, and S. Swiderski. Proving termination by dependency pairs and inductive theorem proving. *J. of Aut. Reasoning*, 47(2):133–160, 2011.

[8] J. Giesl, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski, and R. Thiemann. Proving termination of programs automatically with aprove. In *VSL 2014*, pages 184–191, 2014.

[9] J. Giesl, M. Raffelsieper, P. Schneider-Kamp, S. Swiderski, and R. Thiemann. Automated termination proofs for Haskell by term rewriting. *ACM Trans. Program. Lang. Syst.*, 33(2):7, 2011.

[10] C. Kirchner, R. Kopetz, and P. Moreau. Anti-pattern matching. In *ESOP 2007*, volume 4421 of *Lecture Notes in Computer Science*, pages 110–124, 2007.

[11] M. Korp, C. Sternagel, H. Zankl, and A. Middeldorp. Tyrolean Termination Tool 2. In *RTA '09*, volume 5595 of *LNCS*, pages 295–304. Springer-Verlag, 2009.

[12] A. Krauss. Pattern minimization problems over recursive data types. In *ICFP 2008*, pages 267–274. ACM, 2008.

[13] A. Krauss, C. Sternagel, R. Thiemann, C. Fuhs, and J. Giesl. Termination of Isabelle functions via termination of rewriting. In *ITP 2011*, pages 152–167, 2011.

[14] L. Maranget. Warnings for pattern matching. *J. Funct. Program.*, 17:387–421, 2007.

[15] M. Marin. Extending Mathematica with Anti-Patterns. In *Proceedings of the 12th International Mathematice Symposium*, 2015.

[16] P. Schneider-Kamp, R. Thiemann, E. Annov, M. Codish, and J. Giesl. Proving termination using recursive path orders and SAT solving. In *FroCoS 2007*, pages 267–282, 2007.

[17] R. C. Sekar, R. Ramesh, and I. V. Ramakrishnan. Adaptive pattern matching. In *ICALP '92*, pages 247–260. Springer-Verlag, 1992.

[18] Terese. *Term Rewriting Systems*. Cambridge University Press, 2003. M. Bezem, J. W. Klop and R. de Vrijer, eds.